Debreceni Egyetem

Informatika Kar

Ütközésvizsgálat és válasz a háromdimenziós térben

Témavezető:
Dr. Halász Gábor egyetemi docens
Információ Technológia Tanszék

Készítette:
Konrád Zoltán
Programozó-matemtikus szak

Debrecen

2008

Tartalomjegyzék

Ábrajegyzék

1 Bevezetés

1.1 Az ütközés vizsgálat és válasz célja

Az ütközés vizsgálat célja a 3 dimenziós térben mozgó testek egymással történő ütközésének detektálása és lokalizálása. A lokalizálás azt jelenti, hogy nem csupán felismerjük az ütközés tényét, de megfelelő pontossággal meghatározzuk az ütközésben részt vevő testek felületének ütköző részeit. Ez az ütközés válaszhoz és végeredményben a fizikai szimulációhoz szükséges.

Az üközés válasz két test ütközésekor fellépő mozgásállapot változások számítását jelenti.

Cél a valós idejű fizikai szimulációhoz szükséges gyors kereső algoritmus írása, és a nagyobb sebességet szolgáló adatszerkezet felépítése.

1.2 Az ütközés vizsgálat problémája

A háromdimenziós számítógépes grafikában a testek határoló felületét poligon-hálóval közelítik. Ezek a poligonok szinte kivétel nélkül háromszögek, és a testek ábrázolására háromszögű hálót használnak. Természetes, hogy egy valóságban létező alakzat annál pontosabban modellezhető háromszögű háló segítségével, minél több és kisebb háromszögből áll a modell. Ugyanakkor e modell kirajzolása annál kevesebb időt vesz igénybe, minél kevesebb háromszög alkotja. Testek grafikai modellezése során mindkét szempontot figyelembe kell venni.

Hogy a problémát érzékeltessem, első nekifutásra megpróbálhatjuk a testek háromszögeit páronként vizsgálni egymást metsző háromszögeket keresve. Ha találunk ilyeneket, akkor valamilyen módon megjelöljük azokat. Ez sebesség szempontjából kis poligonszámú modellek esetén elfogadható lenne, de több száz vagy több ezer háromszög esetén elfogadhatatlanul alacsony sebességet eredményezne. A probléma a Descartes-szorzatból adódik, amit két egymásba ágyazott ciklussal lehet megvalósítani. Ha például a két test 500-500 háromszögből áll, akkor ez 250000 vizsgálatot jelent. Ezt ugyan a felére csökkenthetjük,

mivel egy háromszögpárt elegendő egyszer vizsgálni, de ez még így is 125000 vizsgálatot jelent, és tárolnunk kell, hogy mely háromszög párokat vizsgáltuk már meg.
A hatékony ütközésvizsgálat érdekében tehát el kell felejteni ezt a módszert. A megfelelő módszer a testek térbeli felosztása kisebb, hierarchiába rendezett részekre. A szakdolgozatom témája az ütközés vizsgálat egy általam implementált megoldásának, valamint az ütközés felismerése utáni ütközés válasz leírása.

1.3 Célkitűzés

Munkám célja egy olyan statikus könyvtár létrehozása, melynek eszközei lehetővé teszik háromdimenziós alakzatok kezelését a fájlból történő betöltéstől a grafikus megjelenítésen keresztül a fizikai szimulációig. Szakdolgozatom és e dokumentum hangsúlyos része a könyvtár ütközésvizsgálattal és válasszal kapcsolatos algoritmusainak és eszközeinek ismertetése.

2 Technikai háttér

2.1 Használt eszközök

A rendszer döntő része C++ nyelven íródott. Fejlesztői környezetnek a Microsoft Visual Studio 2003-as kiadását használtam Windows XP Professional operációs rendszerben. A téma szempontjából elengedhetetlen volt a grafikus megjelenítés. Mivel ez önmagában is komoly erőforrásokat igényel, kihasználtam a grafikus gyorsító-kártyák előnyét, melyek nem csak a CPU, de az én vállamról is levették a grafikai megjelenítés terhét. Ennek érdekében használtam a Microsoft DirectX Software Development Kit könyvtári eszközeit. (A szakdolgozathoz mellékelt programot a 2008. márciusi SDK-val fordítottam.) A fejlesztés során ismerkedtem meg a DirectX programozói eszközeinek használatával, és mivel nagyon jó hasznát vettem az általa definiált típusoknak és az azokon műveletet végző függvényeknek, ezek szerves részeivé váltak a rendszernek. (Emiatt csak Windows operációs rendszer alatt, DirectX támogatással használható.)

2.2 Programfelépítés

A program, melyet forráskóddal együtt mellékelek valójában négy egységre bontható, és négy Visual Studio projektben van felépítve. Ezek közül a legfontosabb az Object nevű projekt, melynek kimenete egy statikus könyvtár az **Object.lib**. A másik három projekt (bst, test, test2) kimenete egy-egy futtatható állomány, melyek az Object.lib könyvtárat használják, és amelyeket elsősorban tesztprogramként hoztam létre.

2.2.1 Az Object projekt felépítése

- **Fejléc fájlok**
  - Collision.h : deklarálja a Collision osztályt. Ez az osztály végzi az ütközés vizsgálatot és választ. Erről az osztályról bővebben később lesz szó.

- CObject.h : deklarálja az CObject osztályt, és egyéb struktúrákat definiál. Erről az osztályról bővebben később lesz szó.

- **Forrás fájlok**
  - asm.cpp : assembly nyelven írt függvényeket tartalmaz.
  - mymath.cpp : különböző matematikai számításokat végző függvényeket tartalmaz
  - Collision.cpp : az Collision osztály tagfüggvényeinek és más segédfüggvények definíciója, valamit segédváltozók deklarálása.
  - CObject.cpp : az CObject osztály tagfüggvényeinek és más segédfüggvények definíciója valamit segédváltozók deklarálása.

3 Az ütközés vizsgálat módszere

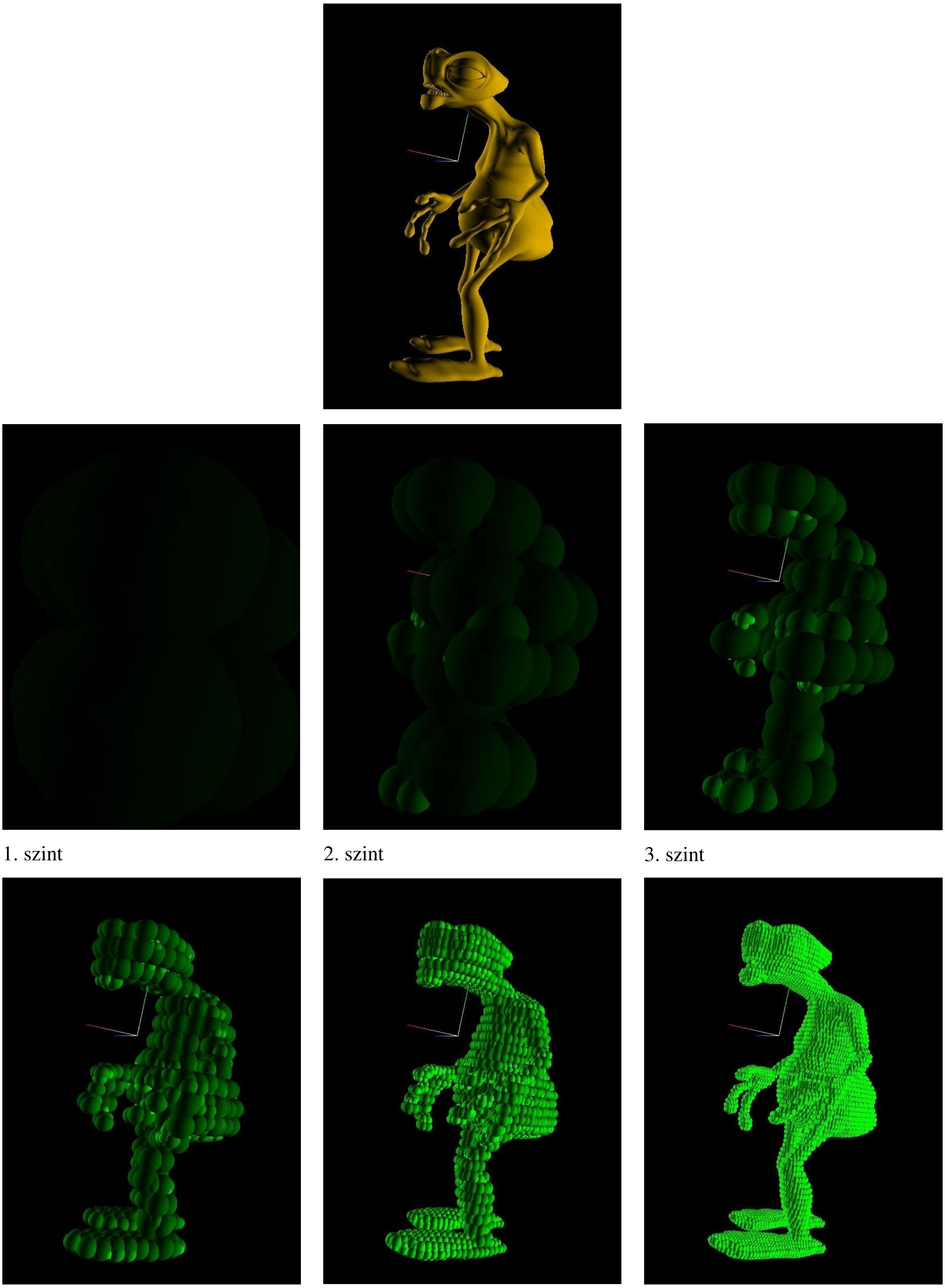
Az alábbiakban ismertetem az általam használt ütközés vizsgálati módszert. Ebben a szakaszban csak az elméleti rész kerül kifejtésre, a gyakorlati megvalósítást külön fejezetben írom le.

A módszer a testet egy fa szerkezetben részekre bontja, és ebben a fában végez keresést. A fa csomópontjai gömbök, melyek a test egy bizonyos részét foglalják be. Minden csomópontnak 8 gyereke van, melyek a szülő által befoglalt részt további nyolc, kisebb részre bontják. A fa gyökere az a gömb, amely a test egészét foglalja be.

A fát felépítő algoritmus a testet befoglaló legkisebb téglatestből indul ki. E téglatest középpontja lesz a fa gyökerét alkotó gömb középpontja, sugara pedig a test középponttól legtávolabb lévő pontjának távolsága. Az algoritmus a téglatestet nyolc kisebb, egyenlő nagyságú téglatestre osztja, melyek közös csúcsa a szülő téglatest középpontja. Ez után megvizsgálja mind a nyolc téglatestre, hogy a test mely pontját tartalmazza illetve, hogy a test melyik háromszögét metszi, és megjegyzi a pontokat, illetve metszéspontokat. Ha nem talál ilyen pontokat, akkor az adott téglatestből nem halad tovább. Ellenkező esetben kiszámolja a pontokat befoglaló gömböt és a téglatestet tovább osztja nyolc kisebbre. Az algoritmus egy előre megadott mélységig építi a fát.

Két test ütközésének vizsgálata során az algoritmus megvizsgálja a testekhez tartozó fák gyökereinek metszését. Ha a gömbök metszik egymást, akkor mindkét fában egy szinttel feljebb lépve az algoritmus újra összehasonlítja a gömböket. Ha valamelyik gömb-pár metszi egymást, akkor az általuk meghatározott részfákon végzi az összehasonlítást. Az algoritmus akkor detektál ütközést, ha talál két olyan egymást metsző gömböt, melyeknek nincsenek gyerekeik. Ha az algoritmus két olyan gömböt talál, melyek nem metszik egymást, akkor az általuk meghatározott részfákon nem halad tovább, mivel abban a testek olyan részei vannak befoglalva melyeknek unióját a részfák gyökerei tartalmazzák, tehát nincs szükség további vizsgálatra. Ez a szerkezet lehetővé teszi a test bizonyos részeinek gyors kizárását a keresésből. Mivel két test ütközése során a testek *nem* ütköző részei a teljes testhez képest nagy számúak, a fák nagyobb részét nem kell bejárni, ezért a vizsgálat gyors.

Ha a gömbfa levélelemeiben lévő gömbök sugarai elég kicsik, az ütközés lokalizálása is jó közelítéssel megvalósul.



1. szint 2. szint 3. szint

4. szint 5. szint 6. szint

1. ábra Egy testre épített 6 szintű gömbfa

4 A gömbfa-építő algoritmus

Az alábbiakban ismertetem a testre gömb-fát építő algoritmust és a hozzá kapcsolódó adatszerkezetet.

4.1 *Adatszerkezet*

A gömbfa a térnyolcadolás technikáját (oct-tree) alkamazza a test hierarchikus részekre bontására. Azaz a fa egy csomópontjának nyolc gyereke van, melyek nyolc részre osztják a csomópont által befoglalt térrészt. Habár a felépített fa csomópontjai gömböket tartalmaznak, az algoritmus téglatestekkel osztja fel a testet, és a test egyes háromszögeit és vertexeit a téglatesthez való viszonya alapján kezeli. A felépített fában azonban ezek a téglatestek már nem szerepelnek.

4.1.1 A csomópontok által befoglalt térrészről

A program kétfajta struktúrával közelíti meg a *csomópontok által befoglalt térrész*t. Az egyik struktúrára csupán az algoritmus futásakor van szükség. Ez egy téglatestet határoz meg; egész egyszerűen tömbökben tárolódnak a téglatest csúcsai, a téglatest lapjainak normálvektorai ill. a téglatest élei. Ezek az adatok az algoritmus futásakor egy háromszög és a téglatest metszésének kiszámolásához szükségesek. Az algoritmus egy a teljes testet befoglaló, legkisebb téglatestből indul ki. Ezt nevezem a testet befoglaló térrésznek. Ezt a térrészt osztja tovább nyolc egyenlő térrészre. A továbbiakban ezeket *térnyolcad*oknak nevezem.

A másik struktúra az, ami a felépített fában ténylegesen megjelenik. Ez írja le a fa csomópontját, ami egy gömb és tárolja az ütközésvizsgálathoz és válaszhoz szükséges adatokat, valamint a fa adatszerkezet implementálásához szükséges gyermekcsomópontokat.

A fát alkotó csomópontokat a NODE nevű struktúra írja le:

```
typedef struct NODE
{
	SPHERE node;
	NODE *children;
```

```
      D3DXVECTOR4 normal;
      int child_num;

}NODE;
```

- node : a csomópont által tartalmazott gömböt leíró struktúra (lásd. SPHERE)
- children : a csomópont gyerekeit tartalmazó NODE tömb
- normal : a gömb által befoglalt háromszögek normálvektorainak az átlaga
- child_num : a csomópont gyerekeinek a száma

A SPHERE struktúra:

```
typedef struct SPHERE
{
      D3DXVECTOR4 center;
      float r;
}SPHERE;
```

- center : a gömb középpontjának koordinátái (a test saját-koordinátarendszerében)
- r : a gömb sugara.

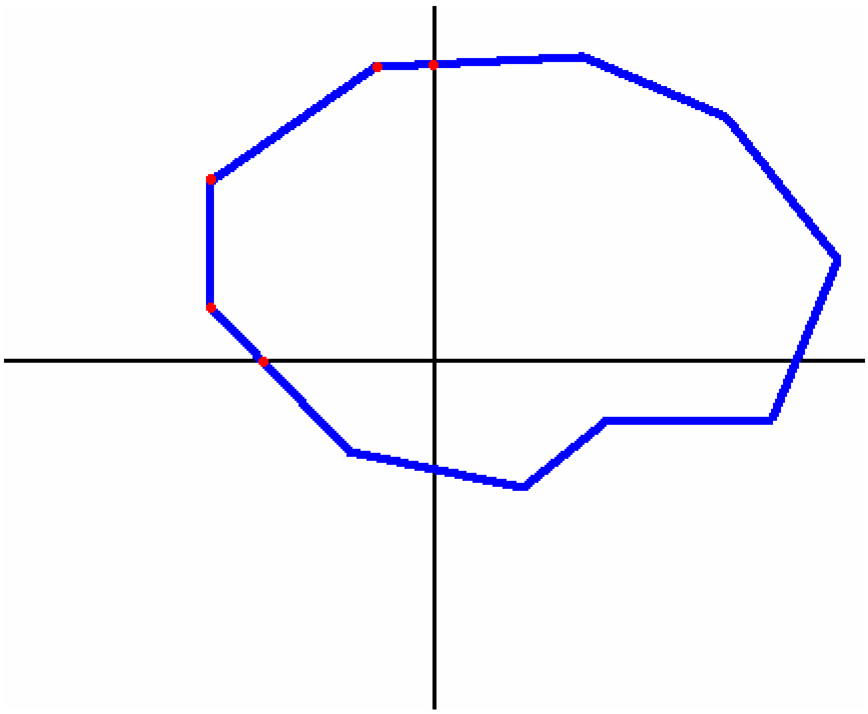
4.2 Az algoritmus

4.2.1 A test csomópontok által befoglalt részének számítása

A fa építése során minden csomópont létrehozásához egy téglatestből indul ki az algoritmus. Ez a téglatest egy térnyolcadot reprezentál. Szükség van a test tér-nyolcad által tartalmazott részeire. Ez a rész a test és a térnyolcadot jelentő téglatest közös része. Az algoritmus a testet alkotó háromszögeken dolgozik, azt vizsgálja, hogy mely háromszögeket metszi a téglatest. Első körben megvizsgálja hogy az adott háromszög teljesen benne van-e a téglatestben. Ha a háromszög minden csúcsa benne van a téglatestben, akkor a háromszög is. Ekkor ezek a csúcspontok bekerülnek a *megtalált pontok listájá*ba. Ha a háromszög valamelyik csúcspontja nincs a téglatestben, akkor megvizsgálja a téglatest élei és a háromszög metszéspontjait. A téglatest egy éle akkor metsz egy háromszöget, ha az él által meghatározott egyenes és a háromszög által meghatározott sík metszéspontja az él egy pontja és a háromszögnek egy belső pontja. Ha ez teljesül, akkor a metszéspont bekerül a *megtalált pontok listájá*ba.

Miután az algoritmus összegyűjtötte a megfelelő csúcspontokat és metszéspontokat, létrehoz egy a pontok köré írható gömböt, mely a fa a csomópontja lesz.

A 2. ábra a síkban ábrázolja a fent leírt csúcspontokat és metszéspontokat. Az ábrán a piros pontok jelölik az adott sík-negyedben megtalált pontokat. A kékkel rajzolt szakaszok felelnek meg a háromdimenziós ábrázolásban alkalmazott háromszögeknek. Látható, hogy a bal felső sík-negyedből kilépő szakaszok (térben a tér-nyolcadból kilépő háromszögek) kívül eső csúcsai nem kerülnek be a megtalált pontok listájába. Ugyanakkor a sík-negyed határát képező él és a kilépő szakaszok metszéspontja bekerül a listába.



2. ábra Egy síkidom egy sík-negyed által behatárolt része

Önmagában a háromszög csúcspontjainak vizsgálata nem elegendő, mert a fában egyre mélyebbre haladva létrejöhetnek olyan kis térnyolcadok, melyek nem tartalmaznak háromszög csúcspontokat, ugyanakkor a test felületét - tehát a test bizonyos háromszögeit - metszik. Továbbá nem jó eljárás, ha a tér-nyolcad által metszett háromszögek csúcspontjait adjuk a megtalált pontok listájához, hiszen ez nem a legszűkebb testfelület részt adja vissza és a fában egyre mélyebbre haladva, nem csökken a térnyolcadok, és ennél fogva a gömbök mérete.

4.2.2 Az algoritmus legfelső szintje

Az algoritmus legfelső szintje vagy külső burka nagyon egyszerű és könnyen leírható pseudonyelven. Az algoritmus mélyebbre nyúló leírása azonban olvashatatlanná tenné a pseodukódot, ezért azt a megoldást választottam, hogy megadom az algoritmus külső héját és a tovább bontandó részeket külön szakaszokban írom le. Ezeket a részeket vastag betűvel és egy római számmal jelölöm, mely alapján azonosítani lehet a rá vonatkozó algoritmus leírását.

1. **Meghatározunk egy** a test egészét befoglaló **téglatestet (I)** ill. **gömböt (II)**. Legyen ez a *gyökérelem*. A gömb lesz a fa gyökéreleme. A téglatest a tér nyolcadolásához szükséges.
2. Hozzuk létre a gyökérelem 8 gyerekét.
3. A *gyökérelem* minden gyerekére ismételjük:
    - 3.1. A *gyerek-csomópont* indexe alapját **létrehozzuk** az általa képviselt **térnyolcadot (III)**.
    - 3.2. Felépítjük a térnyolcadot alkotó *téglatestet*.
    - 3.3. A testet alkotó minden *háromszögre* ismételjük:
        - 3.3.1. Ha a *háromszög* minden csúcsa benne van a *téglatest*ben, akkor maga a *háromszög* is, ezért a csúcspontokat hozzáfűzzük a listához és lépünk a következő *háromszög*re.
        - 3.3.2. Egyébként a *téglatest* minden *él*ére ismételjük:
            - 3.3.2.1. **Ha az *él* metszi a *háromszög*et (IV)**, akkor a metszéspontot hozzáfűzzük a listához.
    - 3.4. Ha a lista elemeinek száma nulla, akkor
        - 3.4.1. a *gyerek csomópon*t sugara legyen 0
        - 3.4.2. a *gyerek csomópont* gyerekeinek a száma legyen 0
    - 3.5. Egyébként hozzuk létre a listában szereplő pontok befoglaló gömbjét és téglatestét.
    - 3.6. Ha a mélységkorlát > 0 és a lista elemeinek száma != 0, akkor

3.6.1. Legyen a *gyökérelem* ez a *gyerek-csomópont*.

3.6.2. Vissza a 2. pontra

3.7. Egyébként a *gyerek-csomópont* gyerekeinek a száma legyen 0.

Az algoritmus kiszámolja csomópontokban tárolt normálvektorokat; minden új gömb létrehozásakor veszi a gömb által metszett háromszögek normálvektorainak átlagát, és egységnyi hosszúra állítja. Ez a 6 alfejezetben leírt ütközésválasz megvalósításához szükséges.

4.2.3 A részalgoritmusok:

(I)

Legyen:

minx : a test nulla indexű vertexének x koordinátája

miny : a test nulla indexű vertexének y koordinátája

minz : a test nulla indexű vertexének z koordinátája

maxx : a test nulla indexű vertexének x koordinátája

maxy : a test nulla indexű vertexének y koordinátája

maxz : a test nulla indexű vertexének z koordinátája

1. A testet alkotó minden vertexre ismételjük:
   1.1. Ha *minx* > vertex x koordinatája, akkor *minx* = vertex x koordinatája
   1.2. Ha *miny* > vertex y koordinatája, akkor *miny* = vertex y koordinatája
   1.3. Ha *minz* > vertex z koordinatája, akkor *minz* = vertex z koordinatája
   1.4. Ha *maxx* < vertex x koordinatája, akkor *maxx* = vertex x koordinatája
   1.5. Ha *maxy* < vertex y koordinatája, akkor *maxy* = vertex y koordinatája
   1.6. Ha *maxz* < vertex z koordinatája, akkor *maxz* = vertex z koordinatája
2. Legyen *A*(*minx*, *miny*, *minz*) pont a téglatest egyik csúcsa, *B*(*maxx*, *maxy*, *maxz*) a másik csúcsa. (Ha feltételezzük, hogy a téglatest élei párhuzamosak a kanonikus bázis valamelyik vektorával, akkor az *A* és *B* pont pontosan meghatározza a téglatestet.)

(II)

1. Legyen az (I) algoritmusban meghatározott *A* és *B* pontok által meghatározott szakasz felezőpontja a gömb középpontja, *C*.
2. Legyen $d = 0$
3. A test minden *vertex*ére ismételjük:
   3.1. Ha *C* és a *vertex* távolsága nagyobb, mint d, akkor *d*=*C* és a *vertex* távolsága
4. Legyen a gömb sugara *d*

(III)

Használjuk fel az (I) algoritmus által létrehozott *A* és *B* pontot.

1. Legyen *i* a gyerek-csomópont indexe.
2. Legyen $C = A+((B-A)*0.5)$.
3. Legyen *MIN* és *MAX* a létrehozandó téglalap két, átlós csúcsa.
4. ha $i = 0$, akkor
   4.1. *MIN*.x=*C*.x, *MIN*.y=*C*.y, *MIN*.z=*C*.z, *MAX*.x=*B*.x, *MAX*.y=*B*,y, *MAX*.z=*B*.z.
5. ha $i = 1$, akkor
   5.1. *MIN*.x=*A*.x, *MIN*.y=*C*.y, *MIN*.z=*C*.z, *MAX*.x=*C*.x, *MAX*.y=*B*.y, *MAX*.z=*B*.z.
6. Ha $i = 2$, akkor
   6.1. *MIN*.x=*A*.x, *MIN*.y=*C*.y, *MIN*.z=*A*.z, *MAX*.x=*C*.x, *MAX*.y=*B*.y, *MAX*.z=*C*.z;
7. ha $i = 3$, akkor
   7.1. *MIN*.x=*C*.x, *MIN*.y=*C*.y, *MIN*.z=*A*.z, *MAX*.x=*B*.x, *MAX*.y=*B*.y, *MAX*.z=*C*.z.
8. ha $i = 4$, akkor
   8.1. *MIN*.x=*C*.x, *MIN*.y=*A*.y, *MIN*.z=*C*.z, *MAX*.x=*B*.x, *MAX*.y=*C*.y, *MAX*.z=*B*.z.
9. ha $i = 5$, akkor
   9.1. *MIN*.x=*A*.x, *MIN*.y=*A*.y, *MIN*.z=*C*.z, *MAX*.x=*C*.x, *MAX*.y=*C*.y, *MAX*.z=*B*.z;
10. ha $i = 6$, akkor
    10.1. *MIN*.x=*A*.x, *MIN*.y=*A*.y, *MIN*.z=*A*.z, *MAX*.x=*C*.x, *MAX*.y=*C*.y, *MAX*.z=*C*.z.
11. ha $i = 7$, akkor
    11.1. *MIN*.x=*C*.x, *MIN*.y=*A*.y, *MIN*.z=*A*.z, *MAX*.x=*B*.x, *MAX*.y=*C*.y, *MAX*.z=*C*.z.

(IV)

1. Legyen *t* az egyenes vektori paraméteres egyenletének t paramétere.
2. Kiszámoljuk az *él* által meghatározott egyenes és a *háromszög* által meghatározott sík metszéspontját. A *t* változóba kerüljön a metszéspontot számoló függvény eredménye.
3. Ha t <= 1 és t >= 0, akkor
   - 3.1. Ha a metszéspont benne van a háromszögben, akkor az él metszi a háromszöget.
   - 3.2. Egyébként az él *nem* metszi a háromszöget.
4. Egyébként az él *nem* metszi a háromszöget.

4.2.4 Matematikai számítások

Az (IV) algoritmus 2. pontjában szereplő metszéspont kiszámításánál a sík vektori egyenletét

$$\mathbf{r} \cdot \mathbf{n} + D = 0$$

és az egyenes vektori paraméteres alakját

$$\mathbf{r} = \mathbf{r_0} + t\mathbf{e}$$

használtam fel.

A sík egyenletében szereplő **r** vektor (a sík egy pontjának helyvektora) a háromszög egyik csúcsának a koordinátája, az **n** (a sík normálvektora) pedig a háromszöghöz tartozó normálvektor. Az D paraméter értéke az egyenletből adódik.
Legyen a vizsgált él két végpontja ***a*** és ***b***. Az egyenes egyenletének $\mathbf{r_0}$ vektora (az egyenes egy pontja) a vizsgált téglatest adott élének ***a*** végpontja ($\mathbf{r_0} = \boldsymbol{a}$), az **e** vektor (az egyenessel párhuzamos vektor) az adott él két végpontjából $\mathbf{e} = \boldsymbol{b} - \boldsymbol{a}$ módon adódik.
A metszéspont számítása pedig:

$$(\mathbf{r_0} + t\mathbf{e}) \cdot \mathbf{n} + D = 0, \text{ amiből}$$

$$t = \frac{-D - \mathbf{r0} \cdot \mathbf{n}}{\mathbf{e} \cdot \mathbf{n}}$$

képlettel történik.

A kapott t érték az egyenes vektori paraméteres alakjának t paraméterének egy értéke, a metszéspontot - ami az egyenes egyenletében az **r** – pedig ezen érték behelyettesítésével kapjuk meg.

Ez idáig egy egyenes és egy sík metszetét vizsgáltuk. Az algoritmusnak azonban az adott téglatest egy éle - ami az egyenes pontjainak egy részhalmaza - és az adott háromszög - ami a sík pontjainak egy részhalmaza - metszetére van szüksége. Erről a metszéspontról azonban csak annyit tudunk, hogy az adott háromszög által meghatározott síkon és az adott él által meghatározott egyenesen van, és nem tudjuk, hogy benne van-e az egyenes ill. sík fent említett részhalmazaiban.

Ahogy az (IV) algoritmus 3. pontjában látható, egyszerűen eldönthető, hogy a metszéspont az egyenes pontjainak adott részhalmazában van-e. Ez adódik az egyenes egyenletéből. Az egyenletben szereplő **e** vektor hossza megegyezik a téglatest adott élének hosszával (mivel az **e** vektort $\mathbf{e} = \boldsymbol{b} - \boldsymbol{a}$ módon határoztuk meg). Tudjuk, hogy az egyenes egy adott pontját úgy kapjuk meg, hogy az egyenletben szereplő $\mathbf{r_0}$ pontot eltoljuk az **e** vektor irányába a t paraméterrel megadott arányban. Ha a t paraméter értéke 1, akkor megkapjuk az $\mathbf{r} = \mathbf{r_0} + \mathbf{e}$ pontot, ami az **e** végpontja. Mivel

$$\boldsymbol{a} = \mathbf{r_0} \text{ és}$$
$$\mathbf{e} = \boldsymbol{b} - \boldsymbol{a},$$

az $\boldsymbol{a}$ helyére behelyettesítve az $\mathbf{r_0}$-t és rendezve az egyenletet kapjuk, hogy

$$\boldsymbol{b} = \mathbf{r_0} + \mathbf{e}.$$

Tehát megkapjuk az él $\boldsymbol{b}$ pontját.

Ha t értéke 0, akkor $\mathbf{r} = \mathbf{r_0}$ és mivel $\mathbf{r_0} = \boldsymbol{a}$ megkapjuk az él $\boldsymbol{a}$ pontját. Mivel tudjuk hogy az egyenes minket érdeklő részhalmaza maga az adott él és ez az $\boldsymbol{a}$ és $\boldsymbol{b}$ pontok által meghatározott szakasz, könnyen megadható az egyenes pontjainak általunk keresett részhalmaza, ami:

$$\mathbf{r} = \mathbf{r_0} + t\mathbf{e}, \text{ ahol } t \in \; (]0,1[\subset \mathbf{R}).$$

Ebből adódik a (IV) algoritmus 3. pontjában szereplő feltétel.

Most meg kell határozni a háromszög fent megadott részhalmazát.
Abból indultam ki, hogy ha egy háromszöget felosztunk három kisebb háromszögre, az eredeti háromszög területe megegyezik a három kisebb háromszög területének összegével.
Az (IV) algoritmus 3.1. pontjában szereplő vizsgálatot úgy végzi az algoritmus, hogy a háromszög területét összehasonlítja a metszéspont és a csúcsok által meghatározott három kisebb háromszög területének az összegével. Ha ezek egyenlők, akkor a metszéspont benne van a háromszögben. Ha a három kis háromszög területének az összege nagyobb, mint a háromszög területe, akkor a metszéspont nincs benne a háromszögben. A háromszög területének a számításához a Heron-képletet használtam, amely a háromszög három oldalából számítja ki a területet.

Heron-képlet:

$T^2 = s(s-a)(s-b)(s-c)$, ahol a, b és c a háromszög oldalainak a hossza és $s = (a+b+c) / 2$.

Mivel a számítógép lebegőpontos aritmetikája véges pontosságú, a gyakorlatban a fenti egyenlőség vizsgálat ritkán ad igaz eredményt. Ennek kiküszöbölésére egy hibahatárt állítottam be. Ez a határ *0,01*, mely érték a tesztek során alakult ki.

5 Az ütközés vizsgálat

Az ütközés vizsgálat két alakzat gömbfájának egyidejű bejárásával és az azonos szinten lévő gömbök metszésének vizsgálatával történik. A jelenlegi algoritmus feltételezi, hogy a fák mélysége hat. A fákat egy rekurzív függvény járja be, preorder módon. A függvény paraméterül megkapja a két fa gyökérelemét.

Legyen az A és B alakzat fájának aktuális csomópontja AR és BR, amelyek induláskor a fák gyökerei.

Legyen **ar** és **br** a csomópontokban lévő gömb középpontja.
Legyen **pa** ill. **pb** az A ill. B. test ütközési pontjainak koordinátái.
pa = (0,0,0) és **pb** = (0,0,0).

Legyen **an** ill. **bn** az AR ill. BR. A csomópontokban tárolt normálvektor.
Legyen **na** ill. **nb** az A ill. B. ütköző felületének normálvektora.
na = (0,0,0) és **nb** = (0,0,0).

1. Ha AR-nek vagy BR-nek nincsenek gyerekei, akkor a függvény kilép.
2. Ha AR és BR metszik egymást, akkor
   - 2.1. ha AR és BR a hatodik szinten van, akkor
     - 2.1.1. Legyen **pa** = **pa** + **ar**, **pb** = **pb** + **br**.
     - 2.1.2. Legyen **na** = **na** + **an**, **nb** = **nb** + **bn**.
   - 2.2. Az AR minden gyerekére ismételjük
     - 2.2.1. BR minden gyerekére ismételjük
       - 2.2.1.1. AR legyen az AR aktuális gyereke, BR legyen a BR aktuális gyereke
       - 2.2.1.2. Rekurzívan lépjünk az 1. pontra.

Az algoritmus, akkor detektál ütközést, ha talált legalább egy metszésben lévő gömbpárt a fa hatodik szintjén.

Az 2.1.1 és 2.1.2 pontokban az algoritmus összeadja az ütközésben lévő gömbök középpontjait és a csomópontokban tárolt normálvektorokat. A normálvektorok a gömbfa építő algoritmus futása során jönnek létre (lásd 4.2.2 alfejezet).

6 Az ütközés válasz

Szakdolgozatom második része az általam implementált ütközés válasz leírása.

Amikor két test üközésbe kerül, egymásra erőhatást fejtenek ki, és ennek következtében megváltozik mozgásállapotuk. E változás kiszámítását nevezem ütközés válasznak, az angol *collision response* kifejezés alapján.

Az általam írt rendszer ezt a feladatot két szinten végzi el. Először az ütközés detektálásakor a Collision osztály elvégzi azokat a számításokat, amelyek megadják a két testre kifejtett erők nagyságát, irányát és támadási pontját. Ezután a CObject osztály elvégzi a testek mozgásállapot változásának számítását.

A CObject osztály egy példánya egy adott testet jelent és a rá vonatkozó állapotváltozásokat szükségszerűen saját magának kell kiszámolni, mivel *az* ismeri saját fizikai paramétereit és mozgásállapotát. Természetesen egy test állapota külső erők hatására is megváltozhat, de mivel ezek az erők a külső környezet változására jönnek létre, ezeket a test nem ismeri. Pontosabban egy CObject példány nem tudja, hogy a rá ható külső erők hogyan jönnek létre, ugyanakkor hatásukra az ő mozgásállapota változik meg, tehát állapotának változását neki kell számolni.

A külső erőket a környezet szolgáltatja egy CObject példánynak, és ezt a környezetet a Collision osztály képviseli, tehát a külső erők kiszámolása ezen osztály feladata. Ez a magyarázata annak, hogy az ütközés válasz feladatát két szintre osztottam. A következő szakaszban e két szint megvalósításáról lesz szó.

6.1 A test mozgásállapot változásának számítása

Egy merev test térbeli mozgása leírható egy elmozdulással (transzláció) és egy tengely körüli forgással (rotáció). Egy nyugalomban lévő merev testre ható erő hatására a test minden pontja azonos irányban és azonos sebességgel mozdul el (transzláció). Ez a sebesség függ a test tehetetlenségtől is. Továbbá az erő forgatónyomatékának hatására a test pontjai egy tengely körül, azonos szögsebességgel forgó mozgást végeznek (rotáció). A szögsebesség függ a test adott tengelyre vonatkoztatott tehetetlenségi nyomatékától is. E két mozgás összevonásából adódik a test mozgása. Ha a test az erőhatás pillanatában nyugalomban volt, akkor a fent leírt mozgás jön létre. Azonban ha a test az erőhatás pillanatában mozgásban volt, akkor az erő hatására létrejövő mozgásállapot változás függ a test adott pillanatbeli mozgásától is.

Merev testre ható több erő eredőjét redukálhatjuk egy adott támadási pontban ható erőre és egy forgatónyomatékra:

$$\mathbf{F} = \sum F_i \text{ és } \mathbf{M} = \sum M_i.$$

Az általam implementált rendszerben egy CObject példány megkap egy vektorokat tartalmazó listát, amelyek a rá ható erőket jelentik, és egy pontokból álló listát, amelyek az erők támadási pontjának felelnek meg. Az erők hatására létrejövő mozgásállapot változást az alábbi módon számítja ki a CObject osztály.

Kiszámolja a testre ható erők eredőjét (**F**) és az erők által kifejtett forgatónyomatékok összegét (**M**). Továbbá, mivel a különböző erők különböző irányúak, tehát az ezek által kifejtett forgó mozgások tengelyei is különbözőek, kiszámítja e tengelyek (mint a test középpontjára illeszkedő vektorok) eredőjét is (**T**). Végül kiszámolja a test **T** tengelyre vonatkoztatott tehetetlenségi nyomatékát (**Θ**). Ha a test az erőrendszer hatásának pillanatában nyugalomban volt (egyébként ezt az algoritmus külön nem vizsgálja), akkor a kiszámított erő és forgatónyomaték elegendő is a mozgásállapot változás meghatározásához. Ugyanis a transzláció sebessége:

$s = \mathbf{F/m}$, ahol **m** a test tömege vagy tehetetlenségi ereje;

a rotáció szögsebessége pedig:

$$\varphi = \mathbf{M/\Theta}.$$

Ha a test mozgásban volt, akkor további számítások szükségesek. A transzláció mértékét meghatározó erő számítása egyszerű:

$\mathbf{F} = \mathbf{F} + s_a\mathbf{m}$, ahol s_a a test adott pillanatbeli sebessége.

A rotáció szögsebességének számításakor az algoritmus kiszámolja az erőhatás pillanatában meglévő forgási tengely ($\mathbf{T_a}$) és a $\mathbf{T}$ tengely által bezárt szög koszinuszát (α), majd az új szögsebesség:

$\varphi = (\mathbf{M} + \mathbf{M}_a\alpha)/\mathbf{\Theta}$ módon adódik, ahol $\mathbf{M}_a$ a test adott pillanatbeli szögsebességét kiváltó forgatónyomaték.

A szögsebesség egy pozitív érték, ezért az előző és az új szögsebesség összege nem lehet jó eredmény, hiszen minden erőhatáskor növekedne. Továbbá adott tengely körüli forgás irányát a fenti ok miatt a szögsebesség előjele nem határozhatja meg. Ez az adott tengely irányától függ. Ha egy forgási tengelyt ellentettjére állítjuk, akkor a forgás iránya is megfordul, de a szögsebesség nem változik. Ha egy adott tengely körül forgó testre a forgásiránnyal ellentétes erő hat, és ez a szögsebességgel azonos nagyságú szögsebességet idéz elő, akkor a test forgómozgása megáll, illetve ha ugyanez a nagyságú erő a forgási iránnyal megegyező irányú, akkor a szögsebesség megkétszereződik. Tehát szükség van egy előjeles értékre, ami reprezentálja a változás irányát.
Tudjuk, hogy egy hegyesszög koszinusza 0 és 1, egy tompaszög koszinusza pedig 0 és -1 közé esik. A fenti képletben tehát az új szögsebesség növekszik vagy csökken az α értékétől függően, amit pedig a két tengely által bezárt szög határoz meg. Például:

ha $\mathbf{T_a} = -\mathbf{T}$ és $\mathbf{M} = \mathbf{M}_a$, akkor $\alpha = -1$ és
$\varphi = (\mathbf{M} - \mathbf{M}_a)/\mathbf{\Theta} = 0$.

Valamint:

ha $\mathbf{T_a} = \mathbf{T}$ és $\mathbf{M} = \mathbf{M}_a$, akkor $\alpha = 1$ és
$\varphi = (\mathbf{M} + \mathbf{M}_a)/\mathbf{\Theta} = 2\,\mathbf{M}_a/\mathbf{\Theta}$.

Ha a két tengely nem 180 illetve 0 fokos szöget zárnak be, akkor a pillanatnyi szögsebességgel azonos szögsebességet kiváltó erő hatására az eredő szögsebesség nem 0 illetve nem a pillanatnyi kétszerese lesz, tehát függ a két tengely által bezárt szögtől. Minél kisebb hegyesszöget zár be a két tengely, az eredeti szögsebesség annál nagyobb mértékben növeli, és minél nagyobb tompaszöget annál nagyobb mértékben csökkenti az új szögsebesség

értékét. A fenti képletben szereplő α érték ezt is hivatott képviselni, hiszen abszolút értéke hegyesszög esetén annál nagyobb minél kisebb a szög, tompaszög esetén annál nagyobb minél nagyobb a szög.
Végezetül röviden megadom a test adott tengelyre vonatkoztatott tehetetlenségi nyomatékának számolási módját.
A test minden vertexére kiszámolom a tehetetlenségi nyomatékot. Ehhez szükség van az adott vertex tömegére:

$\mathbf{m}_i = \mathbf{m}/n$, ahol n a vertexek darabszáma.

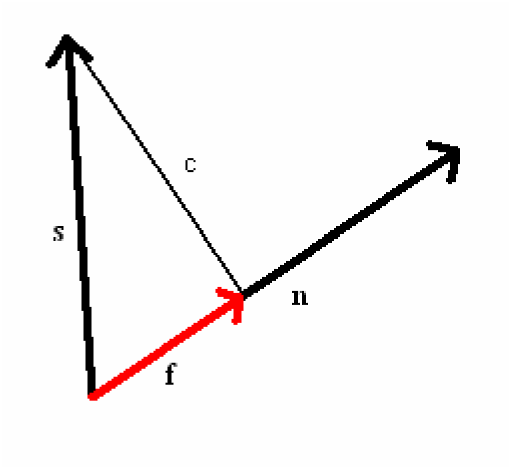
E tehetetlenségi nyomatékok összege adja a test tehetetlenségi nyomatékát.

6.2 A testekre ható külső erők számítása

Az 6.1 alfejezetben leírt számítások a CObject példánynak átadott külső erőrendszer redukálását és az erő hatására kiváltott mozgásállapot változásnak meghatározását végzi. A külső erőrendszer kiszámítását a Collision osztály végzi. Ez az osztály ismeri fel két test ütközésének tényét. Ez az 5 fejezetben leírtak szerint történik.
A 4 fejezetben leírt gömbfa építő algoritmus egy csomópontban letárolja az általa befoglalt háromszögek normálvektorainak egységnyi hosszúra állított eredőjét. Erre az ütközés válasz ezen részében van szükség.

A testre ható külső erők számítása az alábbiak szerint történik.
Az ütközést vizsgáló algoritmus minkét testre külön-külön összeadja az ütközésben részt vevő levél elemekben tárolt normálvektorokat, és a gömbök középpontját. Ez utóbbiakból áll elő a testre ható erő támadáspontja, ill. a test ütközésben részt vevő pontja. Két test ütközésekor tehát egy testnek csak egy pontja vesz részt az üközésben. Ez természetesen nem felel meg a valóságnak, de futási sebesség szempontjából előnyösebb, és a tesztfutások során nem bizonyult túl pontatlannak.
Az algoritmus ezután mindkét testre elvégzi az alábbiakban leírt számítás sorozatot. Most megadom az A testre ható erők számításának módszerét, a B testre ugyanennek a módszernek a tükörképe érvényes. Legyen a két ütközésben lévő test A és B.

Első lépésben kiszámolja a B test ütközésben lévő pontjának a sebesség vektorát. Itt figyelembe veszi a B test sebességét és szögsebességét. Ezután veszi a B testhez fent leírt módon megadott normálvektornak (**n**) és a pont sebességvektora végpontjának (**s**) a távolságát (c) (lásd 3. ábra). Ezután az **s** hosszából és a c-ből Pitagorasz-tétellel kiszámolja az **f** vektor (az ábrán a pirossal rajzolt vektor) hosszát. Az **n** vektort megszorozva ezzel az értékkel és a B test tömegével megkapjuk az A testre ható külső erőt.



3. ábra Külső erő számítása

A számításokban szerepeltetem az üközési számot is, amely az üközés rugalmasságát mondja meg. Ennek értéke 0 és 1 közé esik, 0 érték eseten az ütközés tökéletesen rugalmatlan, 1 érték esetén tökéletesen rugalmas. Az ütközésben résztvevő pont sebesség vektorát (**s**) ezzel az értékkel szorozza az algoritmus.

Miután az algoritmus mindkét testre kiszámolta a külső erőt, befűzi azokat a testekre ható külső erők listájába, majd meghívja a testekre a 6.1 alfejezetben leírt algoritmust implementáló CObject függvényt. Miután a testek kiszámolták saját mozgásukat, ezen algoritmus meghívja a CObject transzlációt és rotációt végző függvényeit.

6.3 Több test egyidejű ütközése

Előfordul olyan eset, hogy egy adott időpillanatban több test ütközik egymással. Ilyenkor egy testre több másik test is erőt fejthet ki. Ennek megoldását a Collision osztály úgy végzi el, hogy a testekre páronként elvégzi a külső erők számítását, de a CObject állapot változást számoló függvényét csak akkor hívja meg, amikor minden ütköző test-párra elvégezte a számításokat. Ilyen esetben egy testre nem csak egy erő hathat, hanem egy erőrendszer, de 6.1 alfejezetben leírt algoritmus erre fel van készítve.

7 Az Object.lib statikus könyvtár rövid ismertetése

Az Object.lib két osztályt tartalmaz. A CObject osztály egy háromdimenziós alakzatot ír le, a Collision osztály tartalmazza az ütközésvizsgálat és válasz függvényeit.

7.1 A CObject osztály

A CObject létrehozásakor olyan osztály megépítése volt a cél, amely tartalmaz egy háromdimenziós alakzat megjelenítéséhez szükséges struktúrát, továbbá tartalmazza azokat a tulajdonságokat és viselkedés elemeket, melyekkel reagálni tud külső fizikai (mechanikai) hatásokra. Egy CObject példány például ismeri saját tömegét, mozgásának irányát és sebességét, képes külső erőhatásra megváltoztatni saját mozgásállapotát, stb.

Ez az osztály tárolja egy háromdimenziós alakzat geometriáját. A tárolási módot úgy alakítottam ki, hogy az megfeleljen a DirectX-szel történő megjelenítésnek. Egy példány megjelenítéséhez többek között szükség van egy DirectX eszköz mutatóra, amelyet a könyvtárat használó programnak kell létrehoznia és átadnia az CObject példánynak.
A CObject továbbá tárolja az adott példányra illeszkedő gömbfát.

Az osztály konstruktorának három változata van:

- `CObject::CObject()` : Ez a konstruktor olyan példányt hoz létre, mely még nem tartalmaz adatokat az objektumról. Az így létrehozott példány geometriájára vonatkozó adatokat fájlból tölthetjük be. Ehhez az osztály tartalmaz egy metódust, melyről a későbbiekben szó lesz. A példány grafikai megjelenítésre nem alkalmas, hiszen a fent említett eszközmutató nincs beállítva. Az így létrehozott egyedre viszont alkalmazhatjuk a gömbfa-építő algoritmust, tehát DirectX támogatás nélkül hozhatjuk létre és menthetjük fájlba egy alakzat fáját.
- `CObject::CObject(LPDIRECT3DDEVICE9,float)` : Ez a változat annyiban különbözik az előzőtől, hogy paraméterül egy eszközmutatót vár (`LPDIRECT3DDEVICE9`). A geometriai adatokat szintén fájlból kell betölteni, de a példány megjeleníthető.

- `CObject::CObject(LPDIRECT3DDEVICE9,VERTEX_N_D3DV*,long,WORD*,long)` : Ezzel a változattal megadhatunk egy vertex-puffert és egy index-puffert. Az így létrehozott példány megjeleníthető. Ilyenkor a hívó programnak kell biztosítania a puffereket.

Az osztály mindenre kiterjedő ismertetése e dolgozatban nem szükséges, ezért a továbbiakban csak azokat a függvényeket ismertetem röviden, amelyek az ütközés vizsgálattal és válasszal kapcsolatosak.

- `void CObject::BuildBSTree(int level)` : Ez végzi a gömb-fa felépítését, a paraméterben megadott mélységig. A 11. oldalon (A gömbfa-építő algoritmus c. fejezet) ismertetett algoritmus implementálása. A függvény az objektum létrehozásakor megadott méretű testre építi a fát, ezért érdemes *egyszeres* méretaránnyal létrehozni az CObject példányt (lásd lejjebb).
- `void CObject::SaveBSTree(char* file)` : Egy már előzőleg felépített gömb-fát a paraméterben megadott fájlba menti.
- `void CObject::LoadBSTree(char* file, float a)` : A paraméterben megadott fájlból betölt egy gömb-fát. A második paraméter egy méretarány. Betöltéskor a gömbök sugarát és középpontját megszorozza ezzel az értékkel, így igazítva az CObject példány méretéhez a fát, ezért egyszeres méretű testre kell meghívni a `BuildBSTree` függvényt.
- `void CObject::ReduceForceSystem()` : A CObject `POINTS` ill. `FORCES` publikus listájába fűzött támadási pontok ill. erők által reprezentált erőrendszert redukálja egy erőre és egy forgatónyomatékra, amiből megadja a test transzlációs és rotációs mozgását.

Az osztály a fenti metódusoknál jóval több függvényt tartalmaz, melyek a megjelenítéstől kezdve a fizikai számításokig többféle feladatot látnak el (például az alakzat transzlációjához és rotációjához szükséges metódusokat), de ezek ismertetése nem tartozik a dolgozat témájához.

Az Object.lib könyvtár ütközésvizsgálattal és válasszal kapcsolatos további elemeit a Collision osztály tartalmazza, amit a követező szakaszban ismertetek.

7.2 A Collision osztály

A Collision osztály nem csupán két CObject példány ütközését vizsgálja, hanem több CObject példányt kezel. Tulajdonképpen azt a virtuális világot jelenti, amelyben CObject példányok vannak jelen, és amelyben ezek egymásra hatást gyakorolnak.
Az osztály statikus metódusokat tartalmaz. Egy, a könyvtárat használó program számára tehát csak egy világ van, minden CObject ebben létezik. Ezeket az osztály egy tömbben tárolja, és tartalmaz olyan metódusokat, amelyek ezt a tömböt kezelik.

Az Collision legfontosabb metódusai:

- `static void Collision::CollisionDetection(float dt)` : E metódus integrálja a szimulációban résztvevő összes alakzat ütközésének vizsgálatát és az ütközésre adott válasz számítását. (Ez utóbbi megvalósításához természetesen a CObject megfelelő metódusait is segítségül hívja.) A paraméterben megadott érték a Δt, ami a fizikai számításokhoz szükséges, valójában a válasz számításakor létrejövő elmozdulás és forgás sebességét ezzel az értékkel szorozza.
- `static bool Collision::Collide_(CObject *A, CObject *B)` : Megmondja, hogy a paraméterül kapott két alakzat ütközésben van-e egymással.
- `static void Collision::CollisionResponse(CObject *A, CObject *B)` : Két, üközésben lévő testre ható erőket számolja ki. A testek mozgásállapot változását maguk a testek számolják ki az e függvény által rájuk rakott erők alapján.

A `CollisionDetection` függvény használja a `Collide_` és a `CollisionResponse` függvényeket, és ezeket minden CObject példányra alkalmazza, majd meghívja a megfelelő CObject tagfüggvényeket. Tehát az Object.lib könyvtárat használó programnak tulajdon képen elegendő e függvényt meghívnia a fizikai szimuláció megvalósításához.

A Collision.cpp forrásfájl tartalmaz egy segédfüggvényt, mely nem tagja az osztálynak:

- `void BSTBejar(NODE* rootA, NODE* rootB, D3DXMATRIX* TMA, D3DXMATRIX* TMB, int collision_l)` : Ez egy rekurzív függvény, mely a paraméterben kapott két gömb-fát járja be egymás metsző gömböket keresve. Ez az 5 fejezetben leírt ütközés vizsgálat implementációja, és a `Collide_` hívja meg.

8 Konklúzió

A gömbfaépítő algoritmus a legtöbb alakzatra jól közelítő fát épít, és ezekre az alakzatokra az ütközés vizsgálat jó pontossággal lokalizálja az ütközést. Az algoritmus tesztelésekor azonban találtam olyan alakzatot, amelyet a rá épített gömbfa nem fedett el teljesen. Ennek következtében az ütközésvizsgálat is helytelenül működött. Ennek javítását még nem végeztem el, de a szakdolgozat szempontjából ennek nincs jelentősége, mert a javítás nem változtat jelentős mértékben a 4 fejezetben leírt algoritmuson.
A jelenlegi rendszerben a gömbfák mélysége 6, ami nagyszámú csúcspontot eredményez, annak ellenére, hogy csak olyan gömböket tesz bele a fába, amelyek tartalmaznak háromszögeket. Egy fát tartalmazó állomány mérete 1 – 5 Mbyte méretű lehet. Így tehát ha egy Object.lib könyvtárat használó program betölt egy fát, akkor a memóriában is ilyen mennyiségű helyet foglal. A hatos mélységet azért választottam, mert ez tűnt a legoptimálisabbnak méret és a vizsgálat pontossága tekintetében.
További hiba, hogy a rendszer ütközésvizsgáló algoritmusa feltételezi, hogy 6 szintű fákon hajtja végre a keresést. Ha egy fa nem hat mélységű, akkor az algoritmus nem detektál ütközést.
Az üközés válasz tekintetében még nem tudtam kidolgozni a megfelelő számítási módot a külső erők kiszámítására. A 6.2 alfejezetben leírt módszer csak közelítőleg számol helyesen.
Az üközés válasz másik eleme, a 6.1 alfejezetben leírt mozgásállapot változás számítása jól sikerült, leszámítva a test adott tengelyre vonatkoztatott tehetetlenségi nyomatékának számítását. A jelenlegi módszerben az $\mathbf{m}_i$ számítása nem a legmegfelelőbb, hiszen a vertexek eloszlása a test felületén nem feltétlenül egyenletes.

Ezen hibák ellenére mondhatom, hogy a gyors ütközésvizsgálat megvalósítása sikerült, és a tesztelések során, kevesebb negatívumot fedeztem fel, mint pozitívumot. Továbbá, sikerült elérnem azt, hogy az ütközés vizsgálat sebessége nem függ az alakzatok részletességétől (a poligonszámtól), hiszen csak a fát veszi figyelembe és nem foglalkozik a testet alkotó háromszögekkel. Ahogy a 1. ábra mutatja, egy alakzatot egy 6 szintű gömbfa levélelemei jó pontossággal befednek, és ez egyben biztosítja az ütközés megfelelő pontosságú lokalizálását is. Ezért úgy érzem elfogadható az a módszer is, hogy a külső erők számításakor a gömbök középpontját veszem az ütközés helyének, mert ez hatékonyság szempontjából nagy előnyt jelent.

Az ütközésvizsgálat e módszere bármilyen bonyolultságú, konkáv alakzatokra is megfelelő gyorsasággal és pontossággal képes az ütközés detektálására és lokalizálására.

Úgy gondolom a test mozgásállapot változásának számítását sikerült jól megoldanom, hiszen az algoritmus képes több külső erő egyidejű hatására valósághű mozgást kialakítani.

Irodalom jegyzék

Könyvek:

Holics László: Fizka 1

Obádovics J. Gyula: Matematika

Nyisztor Károly: Grafika és játékprogramozás DirectX-szel

Internet:

http://isg.cs.tcd.ie/spheretree/

Köszönetnyilvánítás:

Köszönetet szeretnék mondani témavezetőmnek, Dr. Halász Gábornak.